Slides for a Course on the Analysis and Design of Algorithms

Chapter 3: Combinatorial Optimization and the Greedy Method

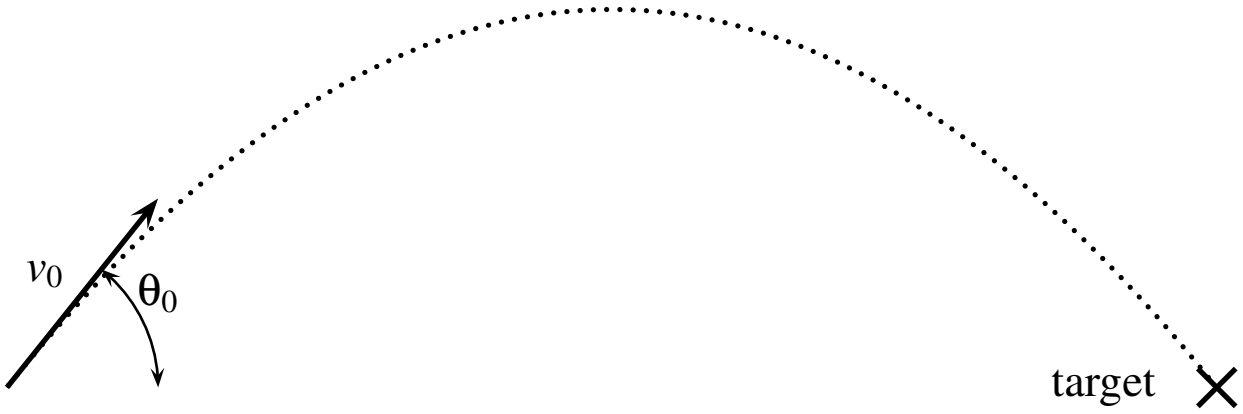
Stephen J. Hegner
Department of Computing Science
Umeå University
Sweden
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

3. Combinatorial Optimization and the Greedy Method

3.1 Properties of Optimization Problems

3.1.1 Nature of Optimization Problems

- Consider the problem, from elementary physics, of firing a projectile so as to hit a given target.



- Two initial values determine the trajectory:
  - v_0 = magnitude of initial velocity of the projectile;
  - θ_0 = angle from horizontal of initial velocity of projectile.
- Each such pair (v_0, θ_0) determines a possible *solution* (landing position), as determine by the laws of physics.
- Solutions may have the following properties:

  admissibility: v_0 does not exceed a specified maximum (based upon available resources).

  feasibility: the projectile hits the target.

  optimality: the total energy expended is a minimum, or the total time required to reach the target is a minimum (as defined in the problem specification).
- The goal here is to carry these ideas over to *discrete* optimization problems.

3.1.2 Knapsack problems – contrasting examples of combinatorial optimization

The common setting:

- A knapsack with weight capacity M.
- n objects $\{\mathsf{obj}_1, \mathsf{obj}_2, \ldots, \mathsf{obj}_n\}$, each with a weight w_i and a value v_i.
- M, the w_i's, and the v_i's are all taken to be positive real numbers.

The simple or continuous knapsack problem:

- Find $(x_1, x_2, \ldots, x_n) \in [0,1]^N$ such that:
  (a) $\sum_{i=1}^{n} x_i \cdot \mathsf{v}_i$ is a maximum, subject to the constraint that
  (b) $\sum_{i=1}^{n} x_i \cdot \mathsf{w}_i \leq M$.

The discrete knapsack problem:

- Find $(x_1, x_2, \ldots, x_n) \in \{0,1\}^N$ such that:
  (a) $\sum_{i=1}^{n} x_i \cdot \mathsf{v}_i$ is a maximum, subject to the constraint that
  (b) $\sum_{i=1}^{n} x_i \cdot \mathsf{w}_i \leq M$.

Note: $[0,1] = \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$.

3.1.3 Example Let $M = 8$; $n = 4$, and let v_i and w_i be as shown in the table below.

i	1	2	3	4
v_i	1	2	5	6
w_i	2	3	4	5

Solution to the simple knapsack problem:

1. Order the objects with greatest $\mathsf{v}_i/\mathsf{w}_i$ first: $\langle 3,4,2,1\rangle$.
2. Add objects, or appropriate fractions thereof, until the knapsack is filled.

   Solution: $\mathsf{obj}_3 + 0.8 \cdot \mathsf{obj}_4$.
   Value: $5 + 0.8 \cdot 6 = 49/5$.

Solution to the discrete knapsack problem:

1. Essentially requires a search of the entire space of possibilities.

   Solution: $\mathsf{obj}_4 + \mathsf{obj}_2$.
   Value: $6 + 2 = 8 = 40/5$.

Note: If the objects are sorted by value per unit weight ($\langle 3,4,2,1\rangle$), with test and select in that order, an optimal solution is not obtained:

obj_3 fits with $\mathsf{v}_3 = 5$, $\mathsf{w}_3 = 4$; include.

obj_4 does not fit; reject.

obj_2 fits with $\mathsf{v}_2 = 2$, $\mathsf{w}_3 = 3$; include.

obj_4 does not fit; reject.

If the objects are sorted by value, with the same strategy, the optimal solution is obtained in this example, but not in general.

3.1.4 Example Let $M = 100$; $n = 3$, and let v_i and w_i be as shown in the table below.

i	1	2	3
v_i	52	50	49
w_i	51	50	50

Solution to the discrete knapsack problem:

- In this case, ordering by nonincreasing $\mathsf{v}_i/\mathsf{w}_i$ is the same as ordering by nonincreasing v_i.
- In each case, the order is $\langle \mathsf{obj}_1, \mathsf{obj}_2, \mathsf{obj}_3 \rangle$.
- A "greedy" strategy of select-and-test of the elements in order does not yield an optimal solution:
  (i) Select obj_1; total value = 52; total weight = 51.
  (ii) Reject obj_2; too heavy.
  (iii) Reject obj_3; too heavy.
- The optimal solution is seen by inspection to be $\{\mathsf{obj}_2, \mathsf{obj}_3\}$, with total value 99.
- Thus, this "greedy" approach does not produce the optimal solution.
- In fact:
  - The best *known* algorithm for the discrete knapsack problem is exponential, although the base may be less than two; *i.e.*, $\Theta(k^n)$ worst-case time complexity for some k with $1 < k \leq 2$. The best *practical* algorithm is $\Theta(2^n)$.
  - The best algorithm for the continuous knapsack problem has $\Theta(n \cdot \log(n))$ worst-case time complexity.

3.2 Matroids and the Greedy Method

3.2.1 Definition – subset systems A *subset system* is a pair $S = (E, \mathfrak{I})$ in which E is a finite set and $\mathfrak{I}$ is a collection of subsets of E which is closed under inclusion. That is, if $A \in \mathfrak{I}$ and $B \subseteq A$, then $B \in \mathfrak{I}$.

The elements of $\mathfrak{I}$ are sometimes called the *independent sets* of S.

3.2.2 Combinatorial Optimization Problems

- A *combinatorial optimization problem* $\mathcal{P}$ is specified by the following:
  (a) A subset system $S = (E, \mathfrak{I})$.
  (b) A profit function $p : E \to \mathbb{N}$.
- A *feasible solution* for $\mathcal{P}$ is any $I \in \mathfrak{I}$.
- An *optimal solution* for $\mathcal{P}$ is a feasible solution J with the property that, for any feasible solution I,

$$\sum_{x \in J} p(x) \geq \sum_{x \in I} p(x)$$

- Sometimes, $p(J)$ is used to denote $\sum_{x \in J} p(x)$.
- Thus, the optimality constraint is expressible as

$$p(J) \geq p(I)$$

3.2.3 Example – the discrete-solution subset system for a knapsack problem Consider the knapsack example of 3.1.3, with $M = 8$; $n = 4$, and v_i and w_i be as shown in the table below.

i	1	2	3	4
v_i	1	2	5	6
w_i	2	3	4	5

The *discrete-solution subset system* is defined as follows.

- The set E is just $\{\mathsf{obj}_1, \mathsf{obj}_2, \mathsf{obj}_3, \mathsf{obj}_4\}$.
- The associated subset system consists of the subsets of $\{\mathsf{obj}_1, \mathsf{obj}_2, \mathsf{obj}_3, \mathsf{obj}_4\}$ which constitute feasible solutions; *i.e.*, which have total weight no greater than 8.
    - The independents sets are thus $\{\mathsf{obj}_1, \mathsf{obj}_2\}$, $\{\mathsf{obj}_1, \mathsf{obj}_3\}$, $\{\mathsf{obj}_1, \mathsf{obj}_4\}$, $\{\mathsf{obj}_2, \mathsf{obj}_3\}$, $\{\mathsf{obj}_2, \mathsf{obj}_4\}$, $\{\mathsf{obj}_1\}$, $\{\mathsf{obj}_2\}$, $\{\mathsf{obj}_3\}$, $\{\mathsf{obj}_4\}$, and $\emptyset$.
- The profit function $p : E \to \mathbb{N}$ just maps a set of objects to the sum of the values of those objects.

3.2.4 Example – the continuous-solution subset system for a knapsack problem Continue with the previous example; $M = 8$; $n = 4$, and v_i and w_i be as shown in the table below.

i	1	2	3	4
v_i	1	2	5	6
w_i	2	3	4	5

The *continuous-solution subset system* is defined as follows.

- Each object of the original problem is divided up into objects of unit weight.
  - obj_1 is divided up into two objects, obj_{11} and obj_{12}, each with weight 1 and value $1/2$.
  - obj_2 is divided up into three objects, obj_{21}, obj_{22}, and obj_{23}, each with weight 1 and value $2/3$.
  - obj_3 is divided up into four objects, obj_{31}, obj_{32}, obj_{33}, and obj_{34}, each with weight 1 and value $5/4$.
  - obj_4 is divided up into five objects, obj_{41}, obj_{42}, obj_{43}, obj_{44}, and obj_{45}, each with weight 1 and value $6/5$.
- The objects of the new system are obj_{11}, obj_{12}, obj_{21}, obj_{22}, obj_{23}, obj_{31}, obj_{32}, obj_{33}, obj_{34}, obj_{41}, obj_{42}, obj_{43}, obj_{44}, obj_{45}.
- The independent sets consist of all subsets of this new set of objects containing at most eight elements.
- The profit function is defined as above, in the obvious way.

3.2.5 The greedy method and matroids The *greedy method* is a general technique for obtaining feasible solutions to an arbitrary combinatorial optimization problem

$$\mathcal{P} = ((E, \Im), p)$$

- The pseudocode is as follows.

$solution \leftarrow \emptyset$;
$pool \leftarrow E$;
while $(pool \neq \emptyset)$ **do**
$\langle$ $e \leftarrow$ member of $pool$ with greatest profit;
$pool \leftarrow pool \setminus \{e\}$;
if $solution \cup \{e\} \in \Im$
then $solution \leftarrow solution \cup \{e\}$;
$\rangle$

- The terminology *greedy* is used because this technique always takes the path of selecting the most profitable local solution.
- As illustrated in 3.1.4, this method does not always yield an optimal solution for a discrete knapsack problem.
- The subset system $S = (E, \Im)$ is called a *matroid* if, for any profit function $p : E \rightarrow \mathbb{N}$, the greedy method finds an optimal solution.
- Sometimes, the term *greedy method* is used more generally for simple select-and-test approaches (such as ordering knapsack objects by profit per unit weight), but in these notes it will be used in the strict sense as defined above.

3.2.6 Fact *Any combinatorial optimization problem associated with an instance of the simple knapsack problem is a matroid.*

PROOF OUTLINE:

- The idea has already been presented in 3.2.4.
- If the capacity M is an integer, just break the objects into pieces of unit weight.
- If the capacity M is not an integer, but is a rational number r/s, multiply it and the weight of each object by s.
    - This results in an equivalent problem with integer capacity and weights.
    - Treat this problem as in 3.2.4.
- If the capacity M is not a rational number, some approximation is necessary.
    - This case is sufficiently esoteric that it will not be considered here.

□

3.2.7 Theorem — characterization of matroids *Let $S = (E, \mathfrak{I})$ be a subset system. Then the following conditions are equivalent:*

(a) *S is a matroid.*

(b) *If $I, J \in \mathfrak{I}$ with $\mathsf{Card}(J) > \mathsf{Card}(I)$, then there is an $e \in J \setminus I$ with $I \cup \{e\} \in \mathfrak{I}$.*
($\mathsf{Card}(I)$ = cardinality of I = number of elements in I.)

(c) *If $A \subseteq E$ and $I, J \in \mathfrak{I}$ are maximal independent subsets of A, then $\mathsf{Card}(I) = \mathsf{Card}(J)$.*
($I \subseteq A$ is a maximal independent subset *of A if for any $K \in \mathfrak{I}$ with $I \subseteq K \subseteq A$, it must be that $K = I$.)*

PROOF:

(a) $\Longrightarrow$ (b): Assume that S is a matroid, and that $\mathsf{Card}(J) > \mathsf{Card}(I)$. Define the following profit function:

$$p(x) = \begin{cases} \mathsf{Card}(J) + 1 & x \in I \\ \mathsf{Card}(J) & x \in J \setminus I \\ 0 & x \notin I \cup J \end{cases}$$

I is suboptimal, since

$$\begin{aligned} p(I) &\leq (\mathsf{Card}(J) + 1) \cdot (\mathsf{Card}(J) - 1) \\ &< (\mathsf{Card}(J))^2 \leq p(J) \end{aligned}$$

The greedy method will start by picking all of the elements if I. Since S is a matroid, the greedy method must yield an optimal solution. This mandates adding some element of $J \setminus I$ to the elements of I which have already been selected, thus establishing the condition.

(b) $\Longrightarrow$ (c): Assume that condition (b) holds, let $A \subseteq E$, and suppose that both I and J are maximal independent subsets of A with $\mathsf{Card}(I) < \mathsf{Card}(J)$. Then (b) guarantees that there is an $e \in J \setminus I$ with the property that $I \cup \{e\} \in \mathfrak{I}$. Clearly, $I \cup \{e\} \subseteq A$ as well, since $e \in J \subseteq A$. This contradicts the maximality of I, whence it must be the case that $\mathsf{Card}(I) = \mathsf{Card}(J)$.

(c) $\Longrightarrow$ (a): Assume that condition (c) holds, then for any fixed $A \subseteq E$, all maximal independent subsets of A have the same cardinality. Let $p : E \to \mathbb{N}$ be any profit function, and let $G = \{g_1, g_2, \ldots, g_m\}$ be any solution obtained by the greedy method. Assume further that $p(g_1) \geq p(g_2) \geq \ldots \geq p(g_m)$. Let $B = \{b_1, b_2, \ldots, b_n\}$ be any maximal independent subset of E for which $p(B) \geq p(G)$. Assume furthermore that $p(b_1) \geq p(b_2) \geq \ldots \geq p(b_n)$. Since G is maximal by construction, it follows that $\mathsf{Card}(G) = \mathsf{Card}(B)$, *i.e.*, $m = n$, by (c) with $A = E$. Furthermore, for all i, $1 \leq i \leq m$, $p(g_i) \geq p(b_i)$. If not, let j be the smallest index for which $p(g_j) < p(b_j)$. Define $A = \{e \in E \mid p(e) \geq p(b_j)\}$. Then $\{g_1, g_2, \ldots, g_{j-1}\}$ is a maximal independent subset of A, since if $\{g_1, g_2, \ldots, g_{j-1}, e\} \in \mathfrak{I}$ with $e \in A$, then since $p(e) \geq p(b_j) > p(g_j)$, the greedy method would have selected e before g_j. However, $\{b_1, b_2, \ldots, b_j\}$ is independent (since it is a subset of an independent set) and a subset of A as well, with $\mathsf{Card}(\{b_1, b_2, \ldots, b_j\}) > \mathsf{Card}(\{g_1, g_2, \ldots, g_{j-1}\})$, contradicting (c). Hence it must be the case that $p(g_j) \geq p(b_j)$, and so G is optimal. $\square$

3.3 Job Scheduling with Unit-Time Jobs

3.3.1 The job scheduling problem

- Given:
  - A set $E = \{\mathsf{job}_1, \mathsf{job}_2, \ldots, \mathsf{job}_n\}$ of *jobs*.
  - A *profit function* $p : E \to \mathbb{N}$.
  - A *deadline function* $d : E \to \mathbb{N}^{>0}$.
- Write $\mathcal{J} = (E, p, d)$.
- Conceptual model:
  - Each job takes unit time to run.
  - Each scheduled job must be completed before its deadline.
  - Only one job may run at a time.
  - Once a job starts, it is not interrupted.
- A *schedule* for a subset $I \subseteq E$ is an injective function

$$f : I \to \mathbb{N}^{>0}$$

  - In schedule f, $f(\mathsf{job}_j)$ is the end time of job_j.
  - Thus, job_j runs from time $f(\mathsf{job}_j) - 1$ to time $f(\mathsf{job}_j)$.
- The schedule f is *legal* if $f(\mathsf{job}_j) \leq d(\mathsf{job}_j)$ for all $\mathsf{job}_j \in I$.
- A subset $I \subseteq E$ is *feasible* if it possesses a legal schedule.
- A subset $I \subseteq E$ is *optimal* if it is feasible, and, for any other feasible $J \subseteq E$,

$$\sum_{\mathsf{job}_j \in J} p(\mathsf{job}_j) \leq \sum_{\mathsf{job}_i \in I} p(\mathsf{job}_i)$$

3.3.2 Example Let a job scheduling problem be defined by the table below.

Job	Profit	Deadline	Time
job_1	100	2	1
job_2	10	3	1
job_3	15	2	1
job_4	27	1	1

- Note that all jobs have unit running time by definition. This value is shown only for emphasis.
- The feasible solutions are all subsets of $\{\mathsf{job}_1, \mathsf{job}_2, \mathsf{job}_3, \mathsf{job}_4\}$ except $\{\mathsf{job}_1, \mathsf{job}_3, \mathsf{job}_4\}$ and $\{\mathsf{job}_1, \mathsf{job}_2, \mathsf{job}_3, \mathsf{job}_4\}$.
- The optimal solution is $\{\mathsf{job}_1, \mathsf{job}_2, \mathsf{job}_4\}$.

3.3.3 The combinatorial optimization problem for job scheduling Let $\mathcal{J} = (E, p, d)$ be a job-scheduling problem with unit-time jobs.

- The *combinatorial optimization problem* associated with $\mathcal{J}$ is just $\mathcal{P} = ((E, \mathfrak{I}), p)$, with

$$\mathfrak{I} = \{I \subseteq E \mid I \text{ is feasible}\}.$$

3.3.4 Lemma *Let $\mathcal{J} = (E, p, d)$ be a job-scheduling problem with unit deadlines, and let $I \subseteq E$. Then I is feasible iff for some (resp. any) ordering $\mathcal{O} = \{\mathsf{job}_{i_1}, \mathsf{job}_{i_2}, \ldots, \mathsf{job}_{i_k}\}$ of I such that*

$$d(\mathsf{job}_{i_1}) \leq d(\mathsf{job}_{i_2}) \leq \ldots \leq d(\mathsf{job}_{i_k}),$$

the function

$$\begin{aligned} f_{\mathcal{O}} : I &\to \mathbb{N}^{>0} \\ \mathsf{job}_{i_j} &\mapsto j \end{aligned}$$

is a legal schedule for I.

PROOF: Let $f : I \to \mathbb{N}^{>0}$ be a legal schedule for I. If there is a pair $(\mathsf{job}_{i_r}, \mathsf{job}_{i_s}) \in I \times I$ such that

$$f(\mathsf{job}_{i_r}) < f(\mathsf{job}_{i_s}) \text{ and } d(\mathsf{job}_{i_r}) > d(\mathsf{job}_{i_s}) \qquad (*)$$

then replace f with $f' : I \to \mathbb{N}^{>0}$ defined by

$$f'(\mathsf{job}_{i_t}) = \begin{cases} i_s & t = r \\ i_r & t = s \\ i_t & \text{otherwise} \end{cases}$$

Then f' is also legal. Repeat this process until there is no pair with the property (*).

The converse is obvious. □

3.3.5 Theorem – job scheduling and matroids *Let $\mathcal{J} = (E, P, d)$ be a job-scheduling problem with unit deadlines, and let $\mathcal{P} = ((E, \Im), p)$ be the associated combinatorial optimization problem. Then $(E, \Im)$ is a matroid.*

PROOF: Let $I, J \in \Im$ with J maximal and $\mathsf{Card}(I) < \mathsf{Card}(J)$, and let

$$f : I \to \mathbb{N}$$
$$g : J \to \mathbb{N}$$

be legal schedules which meet the conditions of 3.3.4. Define $\mathsf{job}_\alpha \in J$ to be the unique element satisfying the following two conditions:

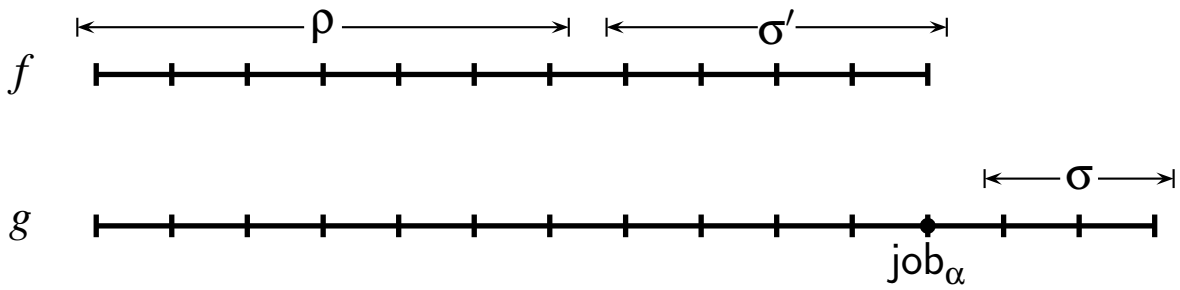
(i) $\mathsf{job}_\alpha \notin I$;

(ii) If $\mathsf{job}_j \in J \setminus I$, then $g(\mathsf{job}_j) \leq g(\mathsf{job}_\alpha)$.

In other words, job_α is the element of $J \setminus I$ which is furthest to the right in g. Since $\mathsf{Card}(I) < \mathsf{Card}(J)$, such a job_α must exist. Now let

$$\sigma = \{\mathsf{job}_j \in J \mid g(\mathsf{job}_\alpha) < g(\mathsf{job}_j)\}$$
$$\sigma' = \{\mathsf{job}_i \in I \mid (\exists \mathsf{job}_j \in \sigma)(f(\mathsf{job}_j) \leq f(\mathsf{job}_i))\}$$

Note that $\sigma \subseteq I$ and that $\sigma \subseteq \sigma'$. σ' defines the shortest final interval of f which contains all elements of σ. These may be visualized as follows, with $\rho = I \setminus \sigma'$.



Now define

$$\sigma'' = \sigma' \setminus \sigma$$

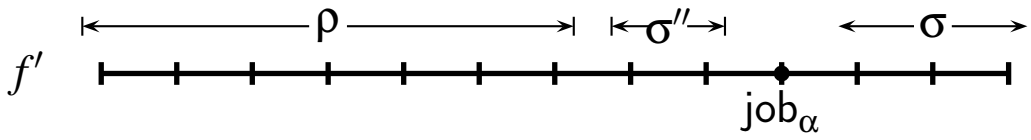
and let

$$f' : I \cup \{\mathsf{job}_\alpha\} \to \mathbb{N}$$

be defined as follows.

- The elements in σ and in σ'' occur in the same order in f' as in f.
- The order of elements in σ' are re-arranged so that all elements of σ'' come before any element of σ; job_α is inserted between these two sets.

This may be visualized as follows.



This schedule is legal because:

- The elements of ρ are not moved.
- The elements of σ'' are scheduled no later in f' than in f.
- The elements of $\{\mathsf{job}_\alpha\} \cup \sigma$ are scheduled no later in f' than in g.

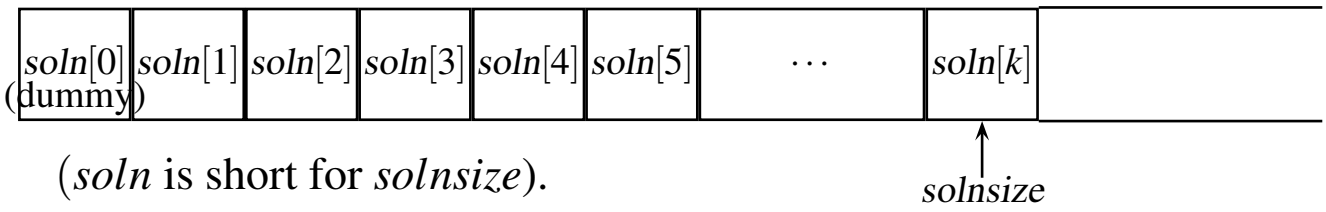
Thus, $I \cup \{\mathsf{job}_\alpha\} \in \mathfrak{I}$, so I Is not maximal in $\mathfrak{I}$. Hence, using the characterization 3.2.7(b), it follows that $(E, \mathfrak{I})$ is a matroid. □

3.3.6 Corollary *The greedy strategy of 3.2.5 always finds an optimal solution of the job scheduling problem with unit deadlines.* □

3.3.7 Basic implementation of job scheduling The following are the key data structures.

Given: *source* : array$[1..n]$ of job; /* Sorted by profit */
Build: *solution* : array$[0..n]$ of job; /* Sorted by deadlline */
Fixed: *solution*$[0]$; /* Dummy job with *deadline* $= 0$ and *profit* $= 0$ */

```
⟨ solution[1] ← source[1];
  solnsize ← 1;
  for i ← 2 to n do
      ⟨ /* A: Find the position for the job source[i]. */
        check ← solnsize;
        while deadline(source[i]) < deadline(solution[check])
                and deadline(solution[check]) > check
              do check ← check − 1;
        /* B: If the job fits, insert it as element check + 1. */
        if deadline(source[i]) > check
          then ⟨ for j ← solnsize downto check + 1 do
                     solution[j + 1] ← solution[j];
                 solution[check + 1] ← source[i];
                 solnsize ← solnsize + 1;
               ⟩
      ⟩
⟩
```



(*soln* is short for *solnsize*).

3.3.8 Proposition *The time complexity of the job scheduling algorithm of 3.3.7 is:*

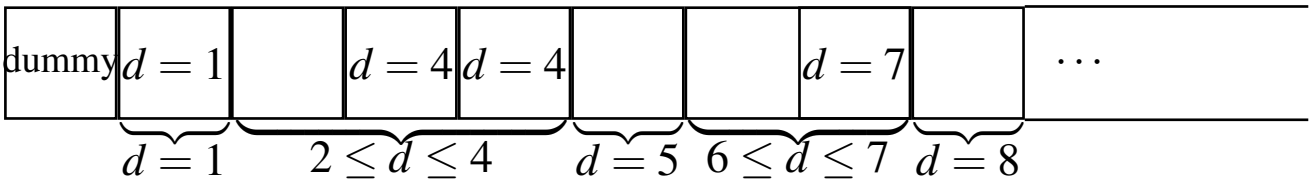
- $\Theta(n^2)$ *in the worst case;*
- $\Theta(n)$ *in the best case;*

plus the amount of time needed to sort the array source*.*

PROOF: The outer loop (A) is executed $n-1$ times, regardless of the nature of the input. The inner loop (B) is executed once in the best case (that the new job is inserted at the right of the existing list) and i times in the worst case (that the new job is inserted at the left of the existing list). This easily accounts for both cases. □

3.3.9 Improving upon the time complexity

- The key idea is to put each new job in the rightmost possible slot.



- The slots are grouped, with one vacant slot per group (except possibly for the first group).
- The vacant slot is leftmost in the group.
- A new element with deadline d in the indicated range may be inserted into the group. It is then combined with its neighbor to the left.
- Effective management of this situation calls for the *partition-union data type*.

3.3.10 The partition-union data type Let S be a finite set. The *partition union data type* on S is structured as follows.

values: partitions of the set S;

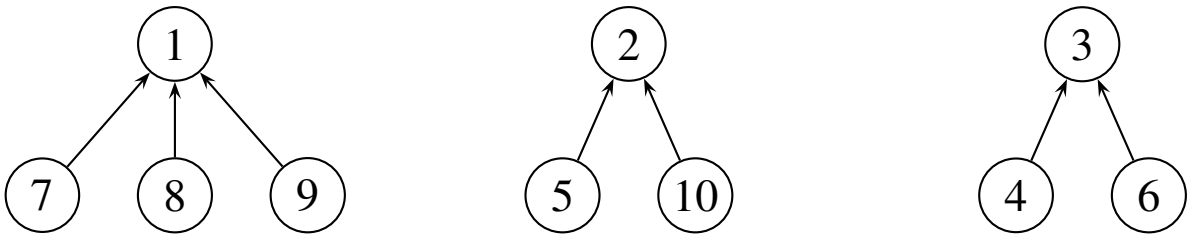
operations:

find: Determine whether or not two elements of S belong to the same block of the partition;

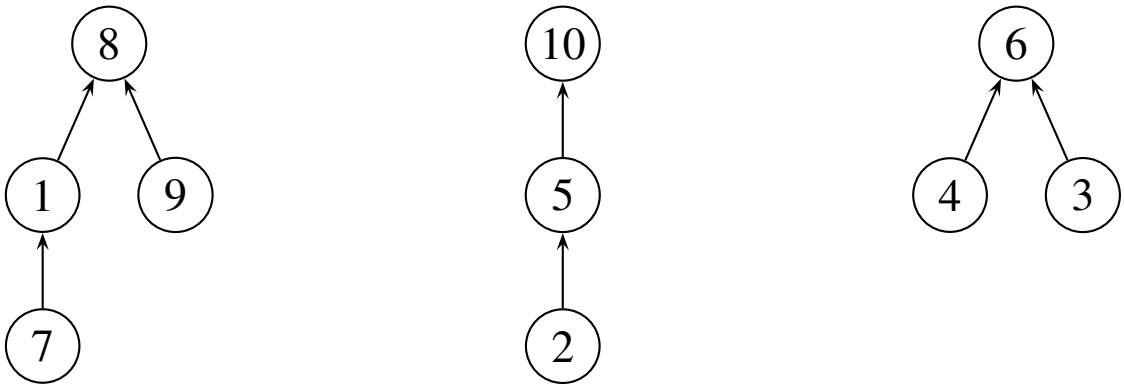
union: Combine two blocks of the partition into one.

3.3.11 An effective realization using rooted trees

- The illustration is via example. Let
  - $S = \{1, 2, \ldots, 10\}$;
  - Partition $= \{\{1,7,8,9\},\{2,5,10\},\{3,4,6\}\}$.
- A tree representation is as follows.

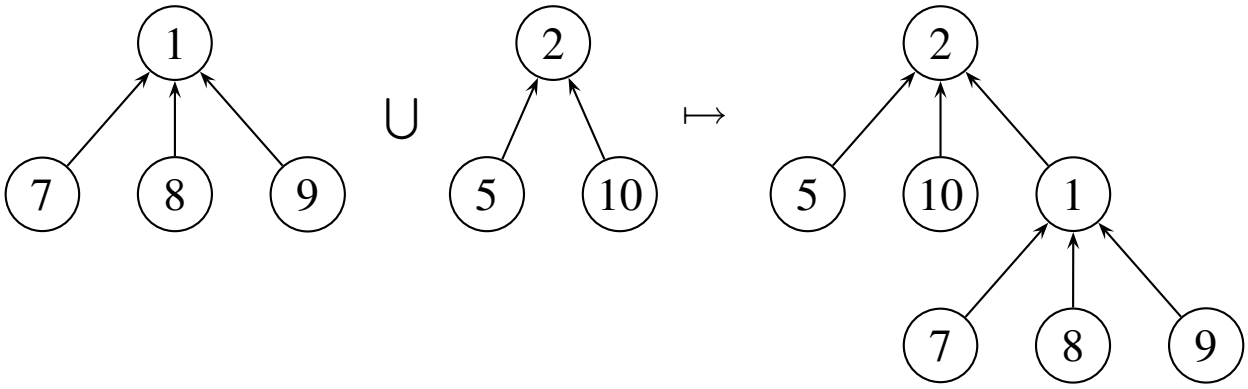


- This representation is not unique; the only requirement is that each block of the partition be represented by a rooted tree.

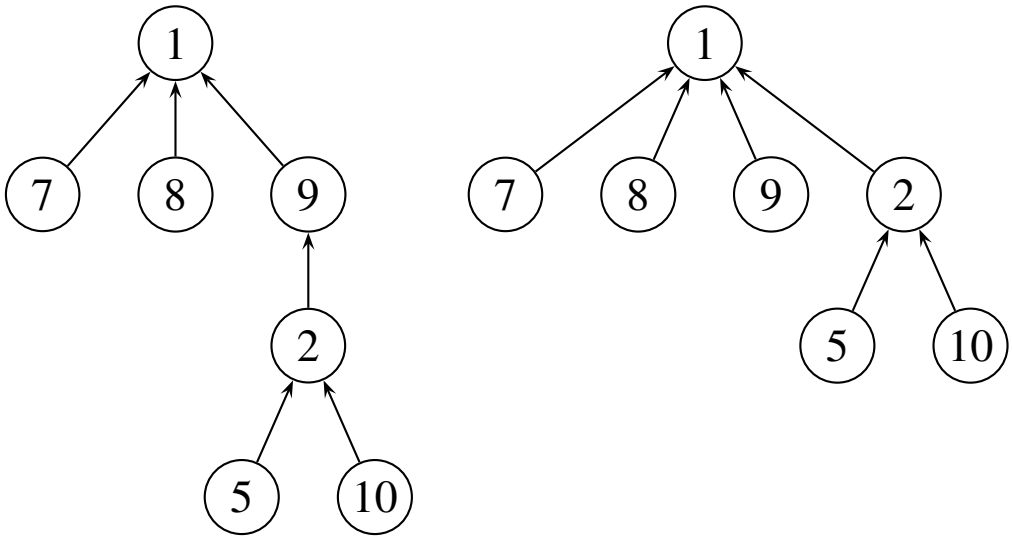


- To implement the find operation, simply follow the path from each vertex to its root, and compare.

- To implement the union operation, glue the trees together.



- This gluing is not unique; other possibilities include the following.

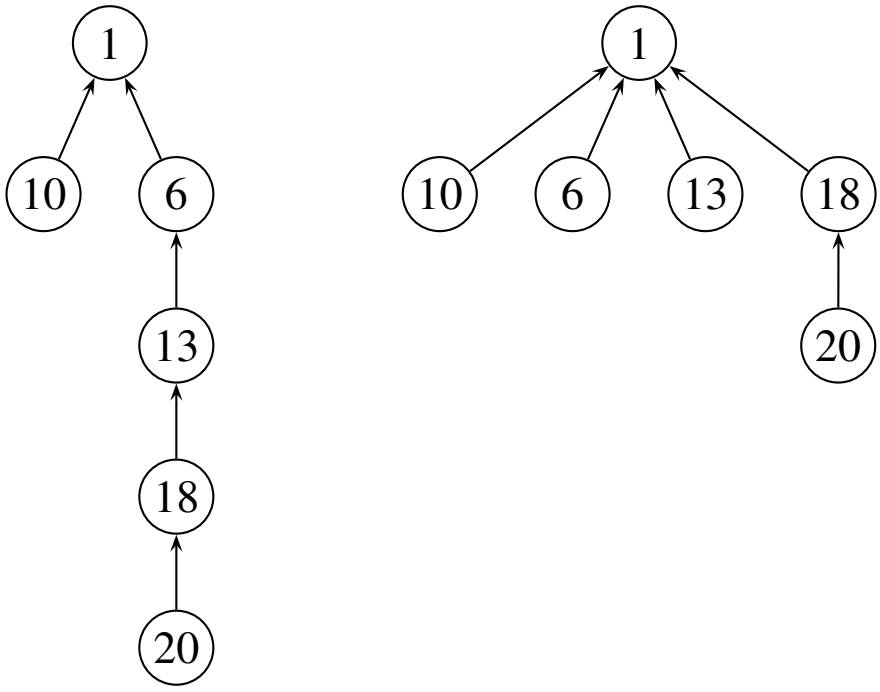


- To have fast finds, it is essential to keep the depth of the trees to a minimum.
- Two rules to help achieve this goal are the following.

  Weighting rule: In the union operation, glue the tree with fewer vertices to the root of the other one.

  Collapsing rule: In a find, move all vertices which are examined to a direct connection to the root.

- In a find of 18 in the tree to the left, the tree to the right is obtained.



3.3.12 Theorem (Tarjan) *In the context of a partition-union data structure, assume that the following operations may be performed in unit time.*

(a) *Following a pointer one level up.*

(b) *Assigning a new value to a pointer.*

Then the worst-case time complexity for any mix of m finds and $n-1$ unions, with $m \geq n$, is $O(m \cdot \alpha(m,n))$, in which $\alpha(m,n)$ grows extremely slowly. ($\alpha(m,n) \leq 5$ for any practical case.) Thus, for all practical purposes, the computation is linear in m.

PROOF: Consult the following references:

[1] Tarjan, R. E., "Complexity of a good but not linear set union algorithm,", *J. ACM* **22**(1975), pp. 215-225.

[2] Tarjan, R. E. and J. van Leeuven, "Worst-case analysis of set union algorithms," *J. ACM*, **31**(1984), pp. 245-281.

□

3.3.13 Corollary *There is an algorithm for the job scheduling problem with unit deadlines which has worst-case time complexity which is almost linear, aside from the complexity of sorting the list of jobs initially.*

PROOF OUTLINE:

- Use the basic data structure of 3.3.9, with each group of slots corresponding to a block in the partition. Each slot contains a field in which the index of a job may be stored.
- Initially, each slot is in a block by itself.
- This includes a block for the dummy slot.
- In each block B, a special value $\mathsf{min}(B)$ identifying the least index within that block is maintained.
- In each block B, each slot execpt for $\mathsf{min}(B)$ is occupied by a job. The slot $\mathsf{min}(B)$ is free.
- To insert a job j with deadline d, find the block B which contains slot d (applying the collapsing rule), put j in slot $\mathsf{min}(B)$, and then merge B with the block which contains $\mathsf{min}(B) - 1$.
- When adjacent blocks are merged, they are combined using the union operation, following the weighting rule.
- Failure occurs when an insertion must be into slot 0.

□

3.4 A Brief Review of Some Key Concepts from Graph Theory

3.4.1 Undirected Graphs An *undirected graph* $G = (V, E, g)$ is given by:

(i) V = finite set of *vertices* or *nodes*;

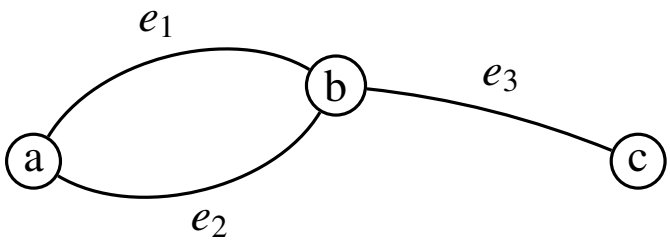
(ii) E = finite set of *edges*;

(iii) $g : E \to \#2(V)$, with $\#2(V)$ denoting those subsets of V containing exactly two elements.

For an example, let

- $V = \{a, b, c\}$
- $E = \{e_1, e_2, e_3\}$
- $g : \quad e_1 \mapsto \{a, b\}, \quad e_2 \mapsto \{a, b\}, \quad e_3 \mapsto \{b, c\}.$

The corresponding graph is shown below.



3.4.2 Directed Graphs A *directed graph* $G = (V, E, g)$ is given by:

(i) V = finite set of *vertices* or *nodes*;

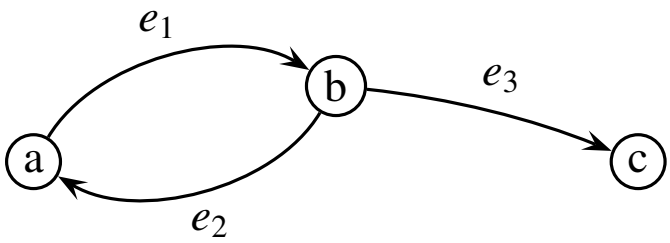
(ii) E = finite set of *edges*;

(iii) $g : E \to V \times V$.

For an example, let

- $V = \{a, b, c\}$
- $E = \{e_1, e_2, e_3\}$
- $g : \quad e_1 \mapsto (a, b), \quad e_2 \mapsto (b, a), \quad e_3 \mapsto (b, c).$

The corresponding graph is shown below.



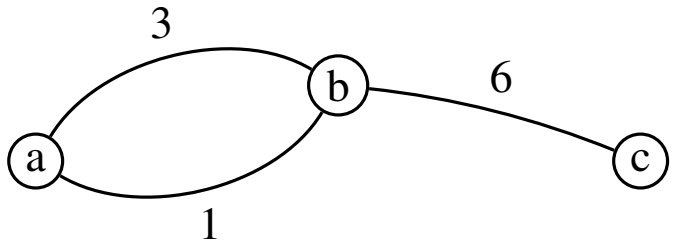
- The term "graph" will be used to denote both directed and undirected graphs, when the type does not matter or may be determined from context.

3.4.3 Labellings and subgraphs Let $G = (V, E, g)$ be a graph (either directed or undirected).

(a) Let S be any set. An *S-labelling* for G is any function $f : E \to S$.

- The following illustrates an example for the undirected graph of 3.4.1.

$$\begin{aligned} S &= \mathbb{N} \\ f(e_1) &= 3 \\ f(e_2) &= 1 \\ f(e_3) &= 6 \end{aligned}$$



(b) A *subgraph* of $G = (V, E, g)$ is any graph $G' = (V', E', g')$ with

- $V' \subseteq V$
- $E' \subseteq E$
- $g' = g_{|E}$ with $\begin{cases} g'(E') \subseteq \mathbf{2}^{V'} & \text{(undirected graph)} \\ g'(E') \subseteq V' \times V' & \text{(directed graph)} \end{cases}$

(c) A subgraph $G' = (V', E', g')$ of $G = (V, E, g)$ is *full* if each $e' \in E$ with $\left\{ \begin{array}{ll} g(e') \subseteq \mathbf{2}^{V'} & \text{(undirected graph)} \\ g(e') \subseteq V' \times V' & \text{(directed graph)} \end{array} \right\}$ is in E'.

3.4.4 Paths and cycles Let $G = (V, E, g)$ be a graph, and let $u, v \in V$.

(a) A *path* from u to w is a nonempty sequence

$$\langle e_1, e_2, \ldots, e_n \rangle$$

of edges such that there are $v_1, v_2, \ldots, v_{n+1} \in V$ with

(i) $\begin{cases} g(e_i) = \{v_i, v_{i+1}\} & \text{(undirected graph)} \\ g(e_i) = (v_i, v_{i+1}) & \text{(directed graph)} \end{cases}$

(ii) $v_1 = u$ and $v_{n+1} = w$.

(b) In a *simple path*, all vertices are distinct, except possibly for $v_1 = v_{n+1}$.

- In the graphs of 3.4.1 and 3.4.2:
  - $\langle e_1, e_3 \rangle$ is a simple path from a to c.
  - $\langle e_1, e_2, e_1, e_2, e_3 \rangle$ is a path from a to c which is not simple.
- It is possible to speak of an *undirected path* in a directed graph (obvious definition).

(c) A *cycle* in G is a path from a vertex to itself.

(d) A cycle is *nontrivial* if it is a simple path of length greater than one.

(e) An undirected graph is *connected* if there is a path between any two vertices.

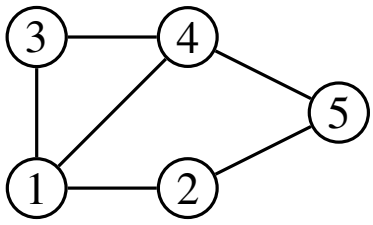
- Usually, a directed graph is taken to be connected if the underlying undirected graph (obtained by forgetting edge direction) is connected.

3.4.5 Trees and spanning trees

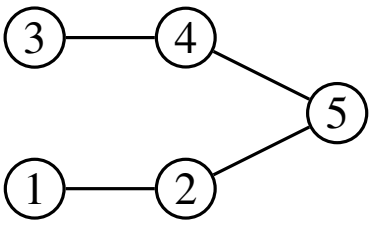
(a) A *tree* in an undirected graph is a connected subgraph containing no nontrivial cycles.

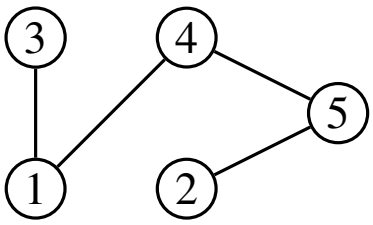
(b) A spanning tree of the graph $G = (V, E, g)$ is a subgraph $T = (V_T, E_T, g_T)$ which is a tree, and which has $V_T = V$.

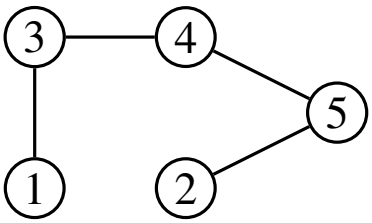
- Let G be the following graph:

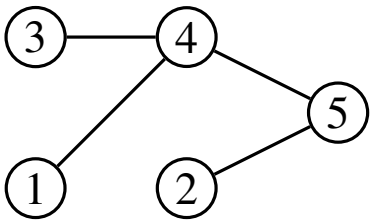


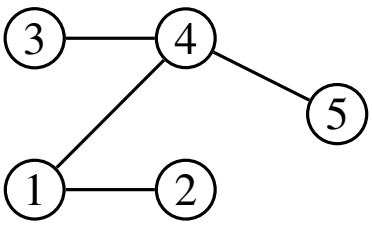
- The following are all examples of spanning trees of G.

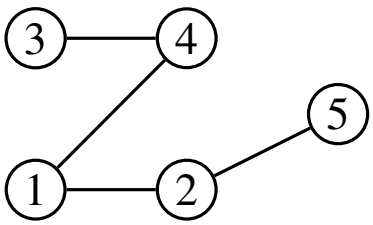












3.4.6 Theorem – characterization of spanning trees *Let $G = (V, E, g)$ be a graph, and let $G' = (V, E', g')$ be a subgraph of G with the same vertices as G. The following conditions are then equivalent:*

(a) *G' is a spanning tree of G.*

(b) *G' is connected and* $\mathsf{Card}(E') = \mathsf{Card}(V) - 1$.

(c) *G' is a tree, but the addition of any additional edge from $E \setminus E'$ results in a cycle.*

PROOF: Easy exercise. □

3.4.7 Corollary *Let G' be a subgraph of the graph G, and suppose further that G' has no cycles. Then G' may be extended to a spanning tree G'' of G.* □

3.5 The Minimum-Spanning-Tree Problem

3.5.1 The minimum-spanning-tree problem

Given: An undirected graph $G = (V, E, g)$ with a labelling

$$p : E \to \mathbb{N}^{>0}$$

Find: A spanning tree $T = (V_T, E_T, G_T)$ of G such that the cost

$$\sum_{e \in E_T} p(e)$$

is a minimum over all such trees of G.

- Typical applications:
  - Linking a communication network with minimum cost.
  - Minimum-cost wiring layout.

3.5.2 Observation *For a connected, undirected graph G with an $\mathbb{N}$-labelling, a minimum spanning tree always exists.*

PROOF: Since G is connected, it must have at least one spanning tree. On the other hand, there are only finitely many spanning trees, so there must be one with minimal cost. □

3.5.3 Convention From now on, all graphs to be considered will have the following properties:

(a) There is at most one edge connecting any two vertices.

(b) No edge connects a vertex to itself.

3.5.4 The matroid of subtrees of a graph Let $G = (V, E, g)$ be any undirected graph.

(a) A *forest* of G is any collection F of subtrees of G with the property that no two distinct members of F have any vertices in common.

- Note that any subgraph of G which contains all vertices of G may be identified uniquely by its set of edges.
- Thus, a forest of G may be viewed as a subset of E with the property that the full subgraph of G defined by those vertices contains no cycles.
- (b) The *graphic matroid* of G is the pair $\mathcal{M}(G) = (E, \mathcal{F}_G)$ with $\mathcal{F}_G \subseteq \mathbf{2}^E$ the set of all subsets of E which define forests of G.

3.5.5 Theorem *Let $G = (V, E, g)$ be a connected, undirected graph. Then $\mathcal{M}(G)$ is indeed a matroid, with the maximal independent sets consisting of precisely the spanning trees.*

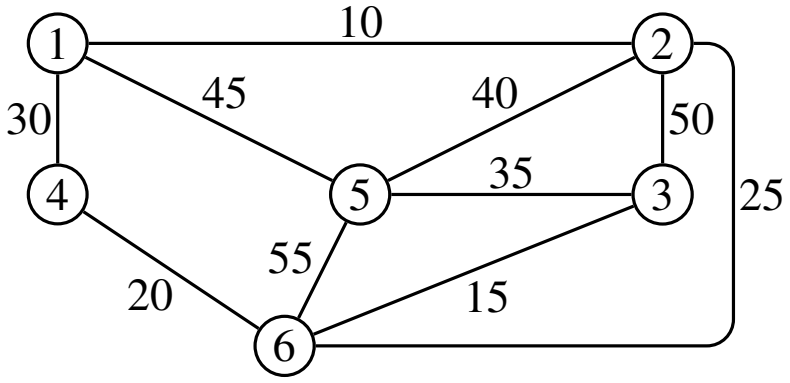
PROOF: The proof follows directly from 3.4.6(b), which asserts that all spanning trees have exactly $\mathsf{Card}(V) - 1$ edges, and that spanning trees are maximal as trees. An application of 3.2.7 completes the proof. □

3.5.6 Kruskal's algorithm for the minimum-spanning-tree problem

- Kruskal's algorithm for the graph $G = (V, E, g)$ with $\mathbb{N}$-labelling $f : E \to \mathbb{N}$ is precisely the greedy algorithm for the associated matroid $\mathcal{F}(G)$.
- Note that this is a *minimization* problem, while the greedy template of 3.2.5 illustrates a *maximization* problem, but there is no essential difference.

$solution \leftarrow \emptyset$;
$pool \leftarrow E$;
while $(pool \neq \emptyset)$ **do**
$\langle$ $e \leftarrow$ edge in of $pool$ with lowest cost;
$pool \leftarrow pool \setminus \{e\}$;
if $solution \cup \{e\} \in \mathcal{F}(G)$
then $solution \leftarrow solution \cup \{e\}$;
$\rangle$

- Example:



Edge Selection:

$\{1,2\}$	10
$\{3,6\}$	15
$\{4,6\}$	20
$\{2,6\}$	25
~~$\{1,4\}$~~	30
$\{3,5\}$	35
~~$\{2,5\}$~~	40
~~$\{1,5\}$~~	45
~~$\{2,3\}$~~	50
~~$\{5,6\}$~~	55

3.5.7 Overview of the implementation of Kruskal's algorithm

1. The solution is constructed as a partition-union data type.
    - The underlying set is the set V of vertices of the graph.
    - The blocks in the partition are collections of vertices which lie in the same tree of the forest.
    - To determine whether a new edge will cause a cycle, it suffices to check whether both of its vertices lie in the same tree of the forest.
        - If they lie in the same tree, adding the edge will create a cycle, so it is not included.
        - If they lie in separate trees, those two trees are merged into a common block upon adding the edge.
2. The algorithm may be stopped when there is only one block remaining in the partition, even if the pool is not empty.
3. To find the next edge of minimum cost in the pool, the data structure known as a *priority queue* is employed.

3.5.8 Priority queues Let $(S, \leq)$ be a finite *ordered* set. The *priority queue data type* on $(S, \leq)$ is structured as follows.

values: subsets of the set S;

operations:

is_empty: Determine whether or not the priority queue is empty.

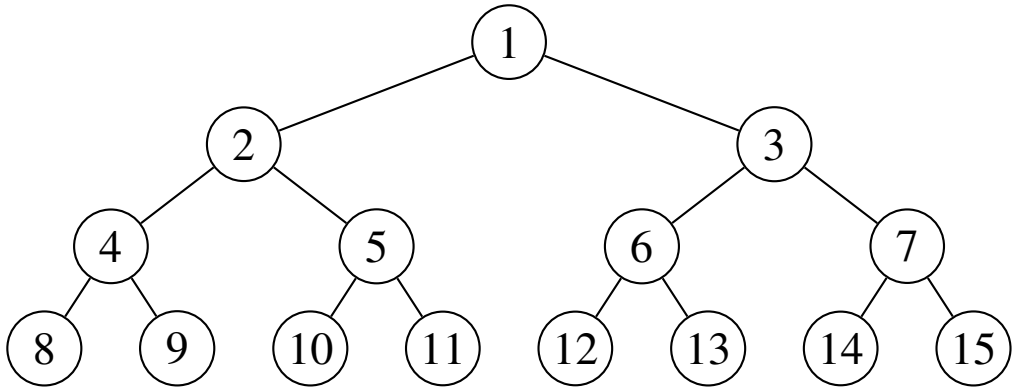
insert_new: Insert a new element into the queue.

retrieve_min: Retrieve the minimum element, and delete if from the queue.

3.5.9 Some important properties of binary trees

(a) A binary tree is *full* if every leaf is the same distance from the root.

(b) The *lexicographical ordering* of the vertices of a full binary tree is breadth first, level-by-level.



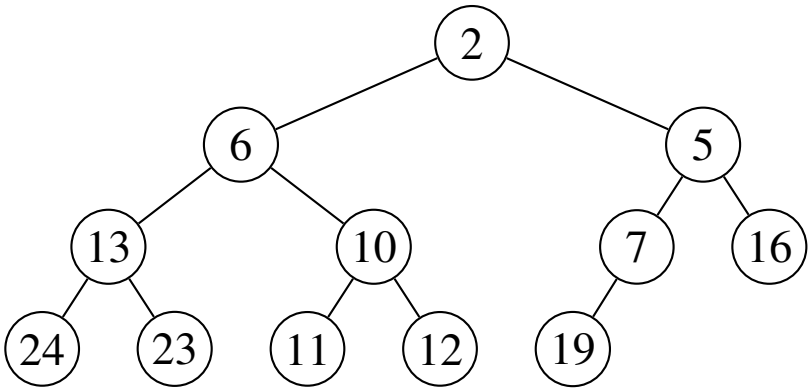
(c) A binary tree is *complete* if it consists of the first k vertices (for $k \in \mathbb{N}$) of some full binary tree.

3.5.10 Realization of priority queues using min-heaps Let T be a complete binary tree with vertices labelled by values in the ordered set $(S, \leq)$.

(a) T is a *min-heap* if, for any vertex t of T,

$$\begin{aligned} \mathsf{Value}(t) &\leq \mathsf{Value}(\mathsf{LeftChild}(t)) \\ \mathsf{Value}(t) &\leq \mathsf{Value}(\mathsf{RightChild}(t)) \end{aligned} \tag{2}$$

- Example:

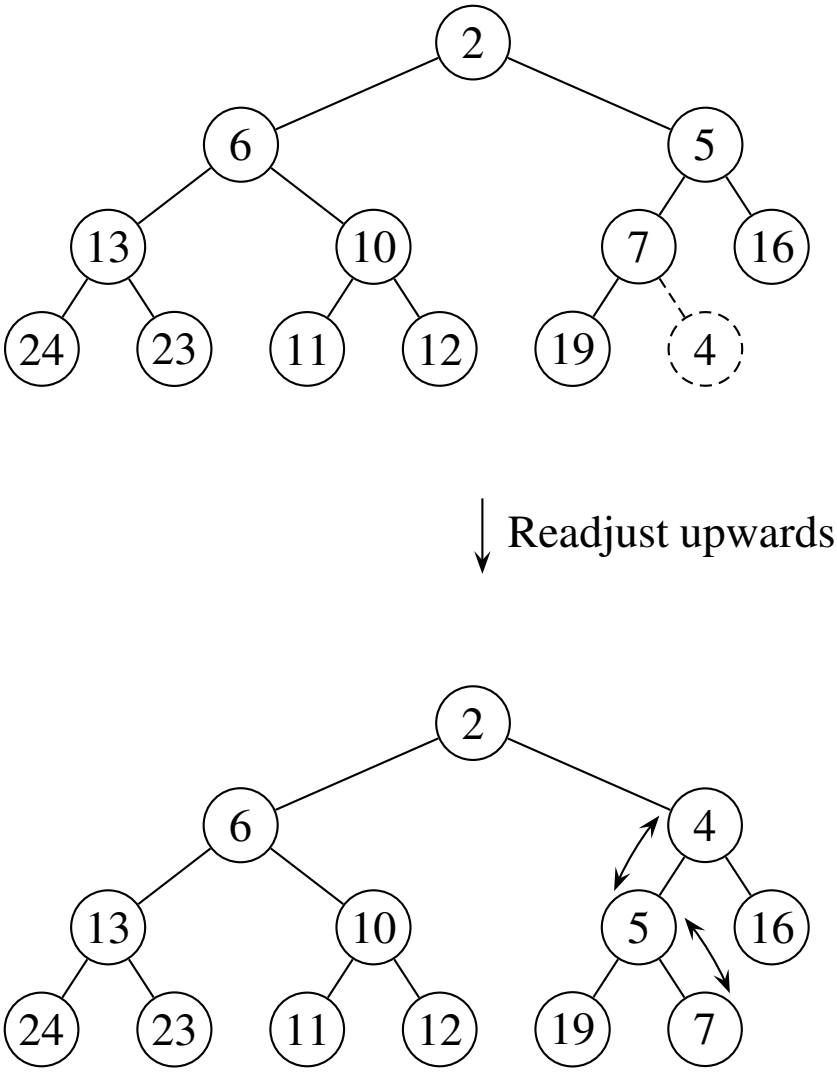


- The implementation of the operators is as follows.

is_empty: Trivial.

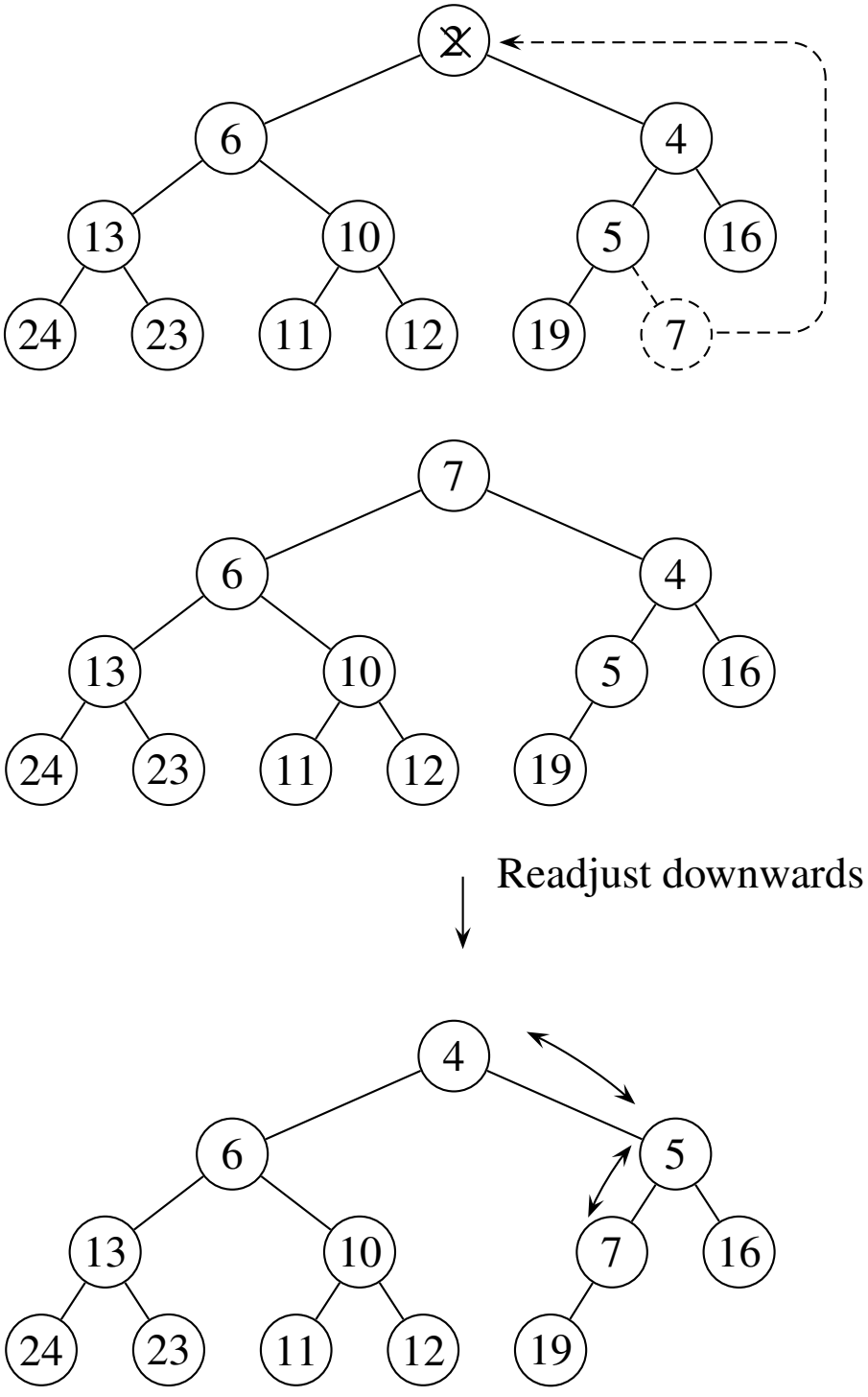
insert_new: Insert the new element as the next element in the complete tree, and then adjust upward.

- Example:



retrieve_min: Delete the root (the minimum element) move the last element in the full-tree ordering up to be the new root, and adjust downward.

- Example:

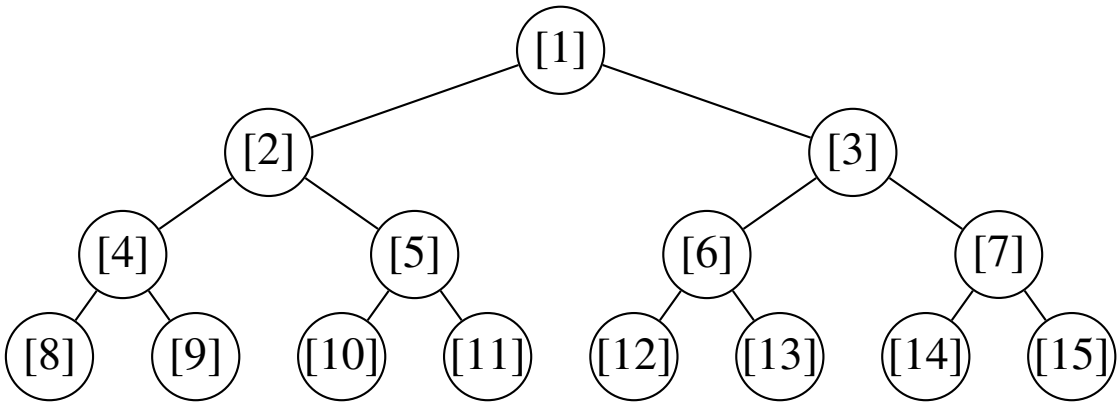


3.5.11 Implementation of complete binary trees using arrays

- The following record type is employed for storage of a complete binary tree in an array.

  compl_bintree = **record**
  tree : array[1..*max_tree_size*] of *value*;
  last : 0..*max_tree_size*;
  end record

- Let T be of type *compl_bintree*. The storage layout is as follows:
  - The root is at $T.tree[1]$.
  - The left child of $T.tree[k]$, if it exists, is at $T.tree[2k]$.
  - The right child of $T.tree[k]$, if it exists, is at $T.tree[2k+1]$.
  - Note that this follows exactly the lexicographic ordering of the vertices of a complete binary tree, as defined in 3.5.9.
  - Such a tree, with the vertices labelled by their position in the array, is shown below.



  - The entry *last* is used to identify the last vertex under row-by-row ordering of the complete binary tree.

3.5.12 The complexity of operations on a min-heap *Let T be a a complete binary tree of size n which is used to realize a min-heap. The complexities of the associated operations are as follows.*

Operation	Complexity	Cases
insert_new	$\Theta(1)$	best
insert_new	$\Theta(1)$	average
insert_new	$\Theta(\log(n))$	worst
retrieve_min	$\Theta(\log(n))$	all
is_empty	$\Theta(1)$	all

PROOF: The $\Theta(1)$ cases. except for the average case of *insert_new*, are immediate. The $\Theta(\log(n))$ cases arise from the need to adjust along a path between the root and a leaf. The average case complexity of inserting a new element is highly nontrivial and is established in the following articles.

- T. Porter and I. Simon, Random insertion into a priority queue structure, *IEEE Trans. on Software Engrg.*, **1**(1975), 292–298.
- E.-E. Doberkat, Inserting a new element into a heap, *BIT*, **21**(1981), 255–269.

□

3.5.13 The "obvious" way to build a min-heap

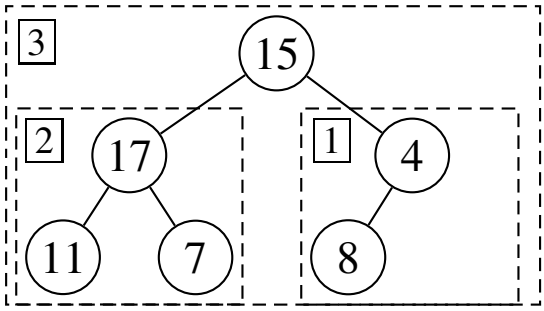
- Repeatedly insert into an initially empty tree.
- The worst-case time complexity for each operation is $\Theta(\log(n))$, for a tree of size n, although the average is $\Theta(1)$ (according to the table of 3.5.12).
- Rough per-opertation complexity argument for the worst case:
  - In total, n elements must be inserted.
  - A typical insertion requires adjustment along half of the tree height.
  - A typical insertion occurs when the tree is half full, resulting in a path length of $n-1$ from root to leaf.
  - Thus, the complexity is

$$0.5 \cdot \Theta(\log(n-1)) = 0.25 \cdot \Theta(\log(n)) = \Theta(\log(n))$$

- Thus, while an average-case complexity of $\Theta(n)$ holds for the entire build operation, the worst-case complexity for this approach will be $\Theta(n \cdot \log(n))$.

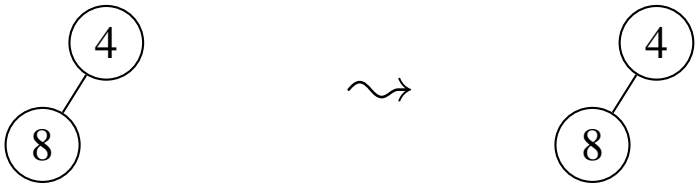
3.5.14 A better way to build a min-heap

- Start with the elements of the tree in arbitrary order, and "heapify" bottom up.
- This results in an amortization of the adjustment complexity, and a consequent reduction in the overall complexity.
- Example:

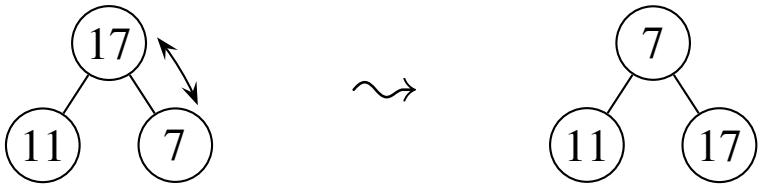


- The subtrees with more than one vertex are processed in the reverse order of the indices of their roots.

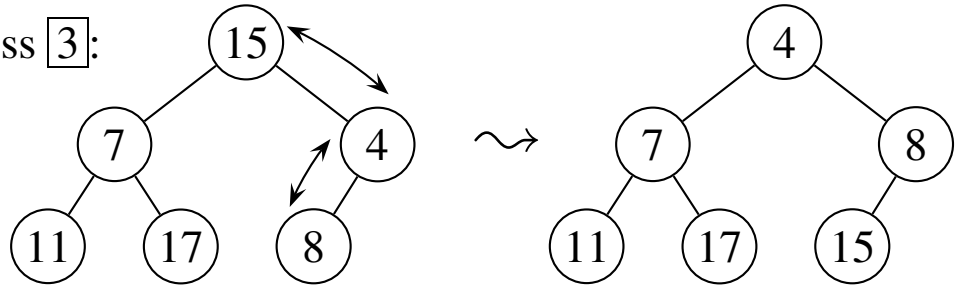
Process 1:



Process 2:



Process 3:



3.5.15 Theorem – the complexity of heapification *The worst-case time complexity of the heapification operation described in 3.5.14 is $\Theta(n)$, with n the number of elements in the tree.*

PROOF:

- Let $k = \lfloor \log_2(n) \rfloor$ = height of the tree.
- Say that the root is at level one.
- The length of a path from a vertex at level i to a leaf is $k - i + 1$.
- Thus, for a vertex at level i, there are $k - i + 1$ swaps in an adjustment, per vertex, maximum.
- The total number of swaps is thus bounded as follows.

$$\sum_{i=1}^{k}(2^{i-1} \cdot (k-i+1)) \;=\; \sum_{i=1}^{k}(2^{k-i} \cdot i) \;=\; 2^k \cdot \sum_{i=1}^{k}(2^{-i} \cdot i) \;\leq\; 2 \cdot n$$

- To see this, note that:

$$\begin{aligned}
2^k &\leq n \text{ and} \\
\sum_{i=1}^{k}(2^{-i} \cdot i) &\leq \frac{1}{2} + (\frac{1}{4} + \frac{1}{4}) + (\frac{1}{8} + \frac{1}{8} + \frac{1}{8}) + (\frac{1}{16} + \frac{1}{16} + \frac{1}{16} + \frac{1}{16}) \\
&\quad + \ldots + (k-1) \cdot (\frac{1}{2^{k-1}}) + k \cdot \frac{1}{2^k} \\
&= \sum_{i=1}^{k} 2^{-i} + \sum_{i=2}^{k} 2^{-i} \ldots + \sum_{i=k-1}^{k} 2^{-i} + \sum_{i=k}^{k} 2^{-i} \\
&\leq 1 + 2^{-1} + 2^{-2} + \ldots + 2^{-(k-1)} + 2^{-k} \\
&\leq \sum_{i=0}^{\infty} 2^{-i} = 2
\end{aligned}$$

□

3.5.16 Corollary *The time complexity of the heapification operation described in 3.5.14 is* $\Theta(n)$ *in all cases, with n the number of elements in the tree.* □

3.5.17 Analysis of Kruskal's Algorithm *Let G be a labelled graph representing the minimum-spanning-tree problem, with*

- n_V = *number of vertices;*
- n_E = *number of edges.*

Assume further that $n_E \geq n_V - 1$ (otherwise, in view of 3.4.6, there cannot be a solution). Then Kruskal's algorithm may be realized with a worst-case and average-case running time of $\Theta(n_E \cdot \log(n_E))$, and a best-case time of $\Theta(n_E)$.

PROOF:

- Assume that the graph is specified as a set of edges, with each edge description containing the following:
  - the two endpoints (vertices);
  - the cost
- The vertices are stored in a partition-union data structure.
  - Initially, each vertex is in its own block.
  - When an edge is added, its vertices are combined into a single block.
  - If two vertices lie in the same block, adding an edge connecting them will result in a cycle, so such edges are rejected.

- The high-level pseudocode:

```
1    Build the min heap M of edges, with cost as value;
2    Build the initial partition-union structure N of vertices;
3    solution ← ∅;
4    while (M not empty and N has more than one block) do
5        ⟨ e ← retrieve_min(M);
6          v_1 ← find(N, vertex_1(e));
7          v_2 ← find(N, vertex_2(e));
8          if (v_1 ≠ v_2)
9            then ⟨ union(N, v_1, v_2);
10                  solution ← solution ∪ {e};
11                ⟩
12       ⟩
```

- The test for emptiness of M may be dropped if it is known that the input graph is connected.
- The line-by-line complexity is as follows:

  Line 1: $\Theta(n_E)$ (see 3.5.15).
  Line 2: $\Theta(n_V)$, and since, $n_E \geq n_V - 1$, it is $O(n_E)$ as well.
  Line 3: $\Theta(1)$.
  Line 4: The loop is executed $\Theta(n_E)$ times, average and worst.
  Line 5: $\Theta(\log(n_E))$.
  Lines 6+7: $O(n_E \cdot \alpha(n_V, n_E))$ total for the whole loop (see 3.3.12).
  Line 9: Subsumed by 6+7.
  Line 10: $\Theta(1)$.

- Thus, the loop total (Lines 4-10) is:

$$O(n_E) + \Theta(n_E \cdot \log(n_E)) + O(n_E \cdot \alpha(n_E, n_V)) = \Theta(n_E \cdot \log(n_E))$$

- This complexity is also valid for the entire algorithm, since Lines 1-3 show a lesser complexity.
- These results apply to both the average and worst case.
- The best case is $\Theta(n_E)$. (Exercise) □

3.5.18 Corollary *Kruskal's algorithm may be realized with a worst-case and average-case running time of* $\Theta(n_E \cdot \log(n_V))$*, and a best-case time of* $\Theta(n_E)$*, with* n_E *and* n_V *denoting the number denoting the number of edges and vertices in the graph, respectively, under the constraint that* $n_E \geq n_V - 1$.

PROOF: This follows immediately from 3.5.17, since $\mathsf{Card}(n_E) \leq \mathsf{Card}(n_V)^2$ implies that $\log(n_E) \leq 2 \cdot \log(n_V)$. □

3.5.19 Variations on Kruskal's algorithm

- There is an algorithm which runs in worst-case time $O(n_E \cdot \log(\log(n_V)))$:

  > Yao, A. C., "An $O(|E| \log\log |V|)$ algorithm for finding minimum spanning trees," **4**, pp. 21-23 (1975).

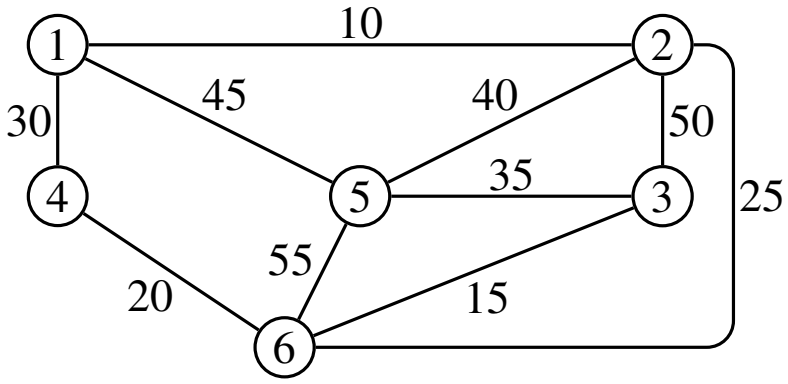
- *Prim's algorithm* (to be examined next).

3.5.20 Prim's Algorithm

- Prim's algorithm differs from Kruskal's in that a single tree (rather than a forest of trees) is maintained during the construction process.
- The high-level pseudocode:

$e \leftarrow$ edge of minimum cost in the set E of all edges;
$solution \leftarrow \{e\}$;
$pool \leftarrow E \setminus \{e\}$;
while $(pool \neq \emptyset)$ **do**
    $\langle$ $e \leftarrow$edge in $pool$ of least cost
        which is connected to some edge in $solution$;
      $pool \leftarrow pool \setminus \{e\}$;
      if $(solution \cup \{e\}$ is a tree$)$
        then $solution \leftarrow solution \cup \{e\}$;
    $\rangle$

- Example:



Edge Selection:

$\{1,2\}$	10
$\{2,6\}$	25
$\{3,6\}$	15
$\{4,6\}$	20
~~$\{1,4\}$~~	30
$\{3,5\}$	35
~~$\{2,5\}$~~	40
~~$\{1,5\}$~~	45
~~$\{2,3\}$~~	50
~~$\{5,6\}$~~	55

Since Prim's algorithm is not formally a greedy algorithm, the results of 3.2.7 do not guarantee that it will always find an optimal solution. A more direct proof is needed.

3.5.21 Theorem — Prim's algorithms finds optimal solutions

Prim's algorithm always finds a minimum spanning tree, whenever such a tree exists.

PROOF: Fix a problem instance for which a spanning tree exists. Let $P = \langle p_1, p_2, \ldots, p_{n-1} \rangle$ be a sequence of edge selections yielded by Prim's algorithm, and let $R = \langle p_1, \ldots, p_k \rangle$ be the maximal initial segment of P which is extendable to an optimal solution. (Surely $k \geq 1$, since Kruskal's algorithm, which is known to be optimal, could select p_1 first.) Now let $Q = \langle p_1, p_2, \ldots, p_k, q_{k+1}, \ldots q_{n-1} \rangle$ be an optimal extension of R. (There is no particular significance to the ordering of the q_i's.) If $k \neq n-1$, then $\{p_1, p_2, \ldots, p_k, q_{k+1}, \ldots, q_{n-1}\} \cup \{p_{k+1}\}$ must contain a nontrivial cycle C. This cycle C must contain p_{k+1} with

- $\mathsf{Cost}(r) < \mathsf{Cost}(p_{k+1})$ for all $r \in C \setminus \{p_1, \ldots, p_{k+1}\}$.

If not, it would be possible to replace some edge $r \in C \setminus \{p_1, \ldots, p_{k+1}\}$ with $\mathsf{Cost}(r) \geq \mathsf{Cost}(p_{k+1})$ by the edge p_{k+1} and maintain an optimal solution, thus contradicting the maximality of R. (Note that any edge in C may be deleted and a spanning tree will result.) However, it must also be the case that

- $\mathsf{Cost}(p_{k+1}) \leq \mathsf{Cost}(r)$ for all $r \in C \setminus \{p_1, \ldots, p_{k+1}\}$ which are adjacent to some member of $\{p_1, \ldots, p_k\}$,

else Prim's algorithm would have selected such an r. Note that there must be at least one such adjacent element, since $\{p_1, p_2, \ldots, p_{k+1}\}$ does not contain a cycle. Thus, no such cycle C can exist, and so $k = n-1$, whence the algorithm yields a minimum spanning tree. □

3.5.22 A basic implementation of Prim's Algorithm

- Prim's algorithm is not amenable to the traditional notion of a min-heap of edges, since the notion of an acceptable candidate edge changes with each addition of a new edge to the tree.
- Rather, the vertices are labelled by members of $\{1, 2, \ldots, n\}$, and the graph is is represented by the array

$$cost = \text{array}[0..n, 0..n] \text{ of integer;}$$

with

$$cost[i, j] = \begin{cases} \text{cost of the edge from } i \text{ to } j & \text{if such an edge exists;} \\ \infty & \text{if no such edge exists.} \end{cases}$$

- Entries for $cost[0, x]$, $cost[x, 0]$, with $x \in 0..n$ are dummies.
- By convention, $cost[i, j] > 0$ for all non-dummy entries.
- As the algorithm proceeds, the following arrays are constructed:

$$\begin{aligned} tree &= \text{array}[1..n-1, 1..2] \text{ of } 1..n; \\ &= \text{array}[1..n-1] \text{ of array}[1..2] \text{ of } 1..n; \\ near &= \text{array}[1..n] \text{ of } 0..n; \end{aligned}$$

with

$tree[i] = (n_1, n_2) = i^{th}$ edge found by the algorithm.

$$near[i] = \begin{cases} \text{index of the vertex in the partial solution} \\ \text{which is nearest (least cost) to vertex } i, \text{ if} \\ \text{vertex } i \text{ is not in the partial solution;} \\ 0, \text{ if vertex } i \text{ is connected to the partial solution.} \end{cases}$$

- The pseudocode for the algorithm:

```
tree[1] ← edge with minimum cost;
for i ← 1 to n do
    if cost[i, tree[1,2]] < cost[i, tree[2,2]]
      then near[i] ← tree[1,2];
       else near[i] ← tree[2,2];
near[tree[1,1]] ← 0;   near[tree[1,2]] ← 0;
min_dist ← 0;   i ← 2;
while (i ≤ n − 1  and  min_dist < ∞) do
      ⟨ for j ← 2 to n do
            ⟨  min_dist ← ∞;
               if (cost[j, near[j]] < min_dist)
                  then ⟨  min_dist ← cost[j, near[j]];
                          min_vertex ← j;
                       ⟩
            ⟩
        if min_dist < ∞
          then ⟨  tree[i] ← (min_vertex, near[min_vertex]);
                  near[min_vertex] ← 0;
                  for j ← 1 to n do
                      if (near[j] ≠ 0 and
                         cost[j, min_vertex] < cost[j, near[j]])
                         then near[j] ← min_vertex;
               ⟩
           else  no spanning tree;
        i ← i + 1;
      ⟩
```

3.5.23 Theorem *The implementation of Prim's algorithm, as given in 3.5.22 has complexity $\Theta(n^2)$ in all cases, with n the number of vertices in the graph.*

PROOF: This is immediate, in view of the matrix representation of the graph. □

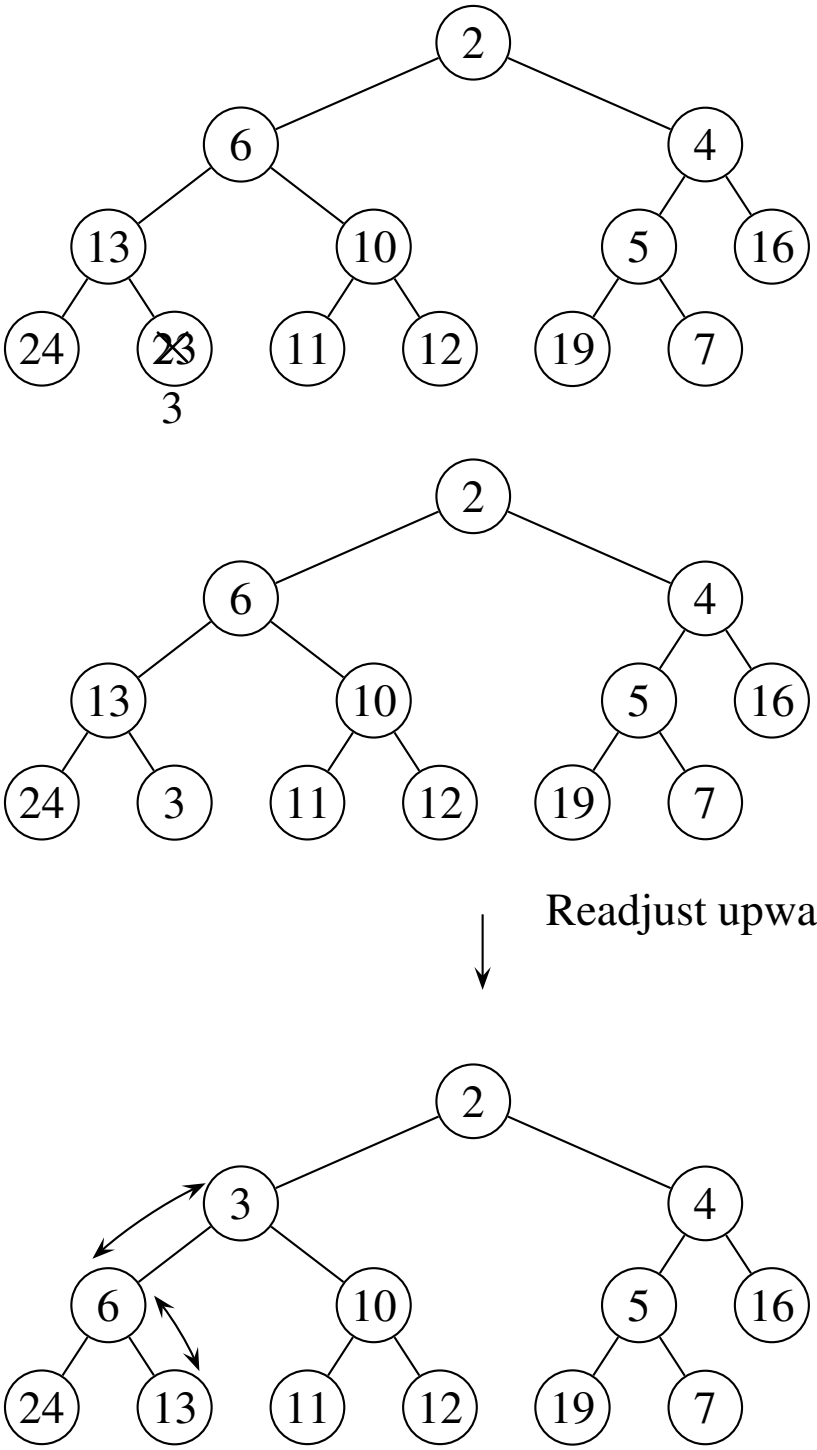
3.5.24 Priority queues with the decrease_elt operation

- Although Prim's algorithm is not amenable to the notion of a traditional min-heap, a modified version in which the weights of objects in the heap may be decreased dynamically is useful.
- Let $(S, \leq)$ be a finite *ordered* set. The *adjustable priority queue data type* on $(S, \leq)$ is structured is an extension of the previously defined priority queue of 3.5.8, with the following additional operation:

  decrease_elt: Replace a particular element in the queue with one of a lesser value.

- The implementation of a priority queue as a min-heap, as described in 3.5.10 may be augmented easily to accommodate this additional operation, by using the readjustment operation as described in 3.5.10.

- Example:


2
6
4
13
10
5
16
24
23
11
12
19
7
3
2
6
4
13
10
5
16
24
3
11
12
19
7
Readjust upwards
2
3
4
6
10
5
16
24
13
11
12
19
7

The following summarizes the operations on this extended version of a min-heap. It is an extension of 3.5.12.

3.5.25 The complexity of operations on an extended min-heap

Let T be a a complete binary tree of size n which is used to realize a min-heap. The complexity of the associated operations is as follows.

Operation	Complexity	Cases
insert_new	$\Theta(1)$	best
insert_new	$\Theta(1)$	average
insert_new	$\Theta(\log(n))$	worst
retrieve_min	$\Theta(\log(n))$	all
decrease_elt	$\Theta(1)$	best
decrease_elt	$\Theta(1)$?	average
decrease_elt	$\Theta(\log(n))$	worst
is_empty	$\Theta(1)$	all

PROOF: All cases except *decrease_elt* were considered in 3.5.12. In the worst case, the complexity of the *decrease_elt* operation is clearly identical to that of *insert_new*. On the average, a vertex in the heap will be only one position away from a leaf, since roughly half of the vertices are at the leaf level, in view of 2.2.11. The best case is immediate. □

3.5.26 An improved implementation of Prim's algorithm

- A key observation is that Prim's algorithm operates on *vertices*, while Kruskal's algorithm operates on *edges*.
- In Prim's algorithm, an adjustable priority queue of *vertices* is maintained.
- The following arrays are used.

$$\mathit{initial\_cost\_to\_tree} = \text{array}[\mathit{vertex}] \text{ of integer}; \quad (3)$$

$$\mathit{nearest} = \text{array}[\mathit{vertex}] \text{ of } \mathit{vertex}; \quad (4)$$

- This array *initial_cost_to_tree* represents the initial minimum cost of an edge from that vertex to the tree under construction; it is used to build the initial priority queue.
- The array *nearest* identifies the vertex in the tree under construction which is nearest to the vertex which is indexed.
- The constant *init_vertex* identifies the initial vertex, which may but not need be connected to a minimum-cost edge.
- For each vertex, a set of those vertices which are adjacent to it in the graph (*i.e.*, connected by single edge) is available.

```
adj_set = array[vertex] of set_of(v_record);
type v_record = record    id : vertex; /* key */
                          dist : pos_integer;
                end;
```

- In the above, *id* is an identifier for the adjacent vertex, and *dist* is the distance from that vertex to the element of the array in which the record occurs.
- Pseudocode is on the next page.

```
1  in_queue[init_vertex] ← false;
2  initial_cost_to_tree[init_vertex] ← 0;
3  foreach v ∈ vertex \ {init_vertex} do
4     ⟨ in_queue[v] ← true;
5       initial_cost_to_tree[v] ← ∞;
6     ⟩
7  foreach x ∈ adj_set[init_vertex] do
8     initial_cost_to_tree[x.id] ← x.dist;
10 /* Build the priority queue M on the set vertex \ {init_vertex} */
11 /* with Priority(x) = initial_cost_to_tree(x). */
12 build_priority_queue(M, vertex \ {init_vertex}, initial_cost_to_tree);
13 while ( not (is_empty(M))) do
14       ⟨ next_vertex ← retrieve_min(M);
15         in_queue[next_vertex] ← false;
16         foreach x ∈ adj_set[next_vertex] do
17           if ((in_queue[x.id] = true)
18                and (x.dist < cost_to_tree[x.id]))
19             then ⟨ nearest[x.id] ← next_vertex;
20                    decrease_elt(M, x.id, x.dist);
21                  ⟩
22       ⟩
```

- The array *in_queue* indicates whether or not a given vertex is in M.
- Upon completion of the algorithm, for each vertex v other than *init_vertex*, *nearest*$[v]$ contains the identity of the vertex which is connected to it via a single edge in the spanning tree.
- Note that the operation *decrease_elt* requires a link from a vertex to its representation in the priority queue.

3.5.27 The complexity of the improved version of Prim's algorithm *Prim's algorithm may be realized with a worst-case running time of* $\Theta(n_E \cdot \log(n_V))$*, an average-case time of* $\Theta(n_V \cdot \log(n_V) + n_E)$*, and a best-case time of* $\Theta(n_E)$*, with* n_E *and* n_V *denoting the number denoting the number of edges and vertices in the graph, respectively.*

PROOF: In the above pseudocode, the loop spanning lines 3-6 takes time $\Theta(n_V)$ in all cases, while the loop spanning lines 7-8 takes $O(n_E)$. Line 12 takes $\Theta(n_V)$ time in all cases. The outer while loop, beginning at line 13, is executed $\Theta(n_V)$ times. Line 14 requires $\Theta(\log(n_V))$ in both the worst and average cases. Note also that the for loop spanning lines 16-21 is executed at most twice for each edge (once for each of its vertices); that is, $2 \cdot n_E$ times. This time is amortized over the entire run, and is not repeated for each iteration of the outer loop. Inside the for loop, only line 20 takes more than constant time; it is bounded by $O(\log(n_V))$ in the worst and $\Theta(1)$ in the average case. Thus, the entire running time is $\Theta(n_V \cdot \log(n_V) + n_E \cdot \log(n_V)) = \Theta(n_E \cdot \log(n_V))$ in the worst case and $\Theta(n_V \cdot \log(n_V) + n_E)$ in the average case.

Although the pseudocode will run in time $\Theta(n_V)$ in the best case (exercise), it will take at least $\Theta(n_E)$ to build the array *adjacent*, so this is a better measure of the best case performance. □

3.5.28 Remarks on Kruskal's vs. Prim's algorithms

- Both algorithms exhibit time complexity $\Theta(n_E \cdot \log(n_V))$ in the worst case. Using the given implementations, Prim's algorithm may be a bit better in the average case.
- Both may be improved to $\Theta(n_E + n_V \cdot \log(n_V))$ in the worst case by replacing the min-heap with a Fibonacci heap. (Not covered in these notes.)
- The choice of which to use is best made via experimental work on appropriate data for the application under consideration.